Address Sensitive Behaviors Causing Benign Divergence

CFAR Technical Report

Thomas Bourguenolle[1], David Poetzsch-Heffter[1], Stijn Volckaert[2], Per Larsen[1], and Michael Franz[1]

[1]UC Irvine, Department of Computer Science
[2]Ghent University, Department of Computer Science

July 2015

1. Motivation

Multi-variant execution environments (MVEEs) are one approach to improve software security [3, 5]. In a MVEE multiple diversified variants of the same program run in parallel. Any difference in behavior between those variants causes the MVEE to abort execution and report the divergence as a possible attack.

Many typical diversifications change the data or code layout [2]. Yet, low-level languages like C allow writing programs whose control flow or data values depend on this layout. These so-called address sensitive programs behave non-deterministically under code and data layout transformations. In a conventional, single-variant execution environment this behavior does not induce problems or is even intended (e.g. for introducing randomness). However, when run in a MVEE, address sensitivity makes the variants diverge resulting in false positives.

In order to make MVEEs compatible with real-world programs, address sensitive behaviors need to be dealt with. As a prior step, this report focuses on giving an overview over existing problematic paradigms. For that, we grouped address sensitive behaviors found in C programs into different categories. Each category comes with a detailed description of the problem in relation to MVEEs and references actual programs that show this behavior. We also give a first indication of the prevalence of the categories in real-world programs and propose approaches to solutions for the individual problems.

2. Categories of address sensitive behaviors

To get a better understanding of address sensitive behaviors we tested various C/C++ programs in a multi-variant execution environment. While most of the linux core utils worked fine, the libX11 and gtk libraries were consequent sources of address sensitivity. Refer to appendix A for a complete list of tested programs. We also created a git repository[1] containing minimal examples that resemble the problematic behavior of each category.

The experiments were carried out using GHUMVEE, which is the most fully featured academic MVEE currently available [5]. The test subjects were compiled using gcc4.8 with the default debug configuration on an Ubuntu 14.04 amd64 system. During the tests we used Address Space Layout Randomization (ASLR, [4]) as a diversification.

In the following, each identified category of address sensitive behavior, is described in detail.

2.1. Use of unintialized value

Common C compilers return the contents of the associated memory cell if an uninitialized variable is used. This value can not be predicted in general and thus creates a source of randomness. In a MVEE this can result in an argument or a syscall mismatch if the control flow of the program depends on the uninitialized value.

Even though the behavior is highly implementation specific (and even undefined in the investigated case) the glibc library seems to apply this idiom on purpose: In the `__gen_tempname()` function (`tempname.c:229`), the variable `static uint64_t value` (line 233) is not initialized and is then used to generate a random temporary file name. The memory at this stack address will almost always be different from one variant to another if ASLR is enabled for instance. Later on, an open syscall with the random file name will trigger an argument mismatch. In another context, one could easily imagine a situation in wich the control flow would be determined by this variable's value and a syscall mismatch would be triggered.

The error could also occur without any data layout diversification but it is less likely since the memory at the variable's address would probably be the same for both variants (due to a recent stack memory free).

It is also worth noting that a similar behavior can result from the use-after-free idiom.

2.2. Treating pointers as integers

Although it is implementation-defined behavior in C, casting pointers to integers is a common practice [1]. A frequent example of this idiom is the use of pointer values as hash keys since it is an easy way to make sure every key is unique. With ASLR enabled, the pointer values being different, one of the hashes could result in a collision and the hashtable would have to be resized while the other variant's would not (see table 1). The control flow would diverge, later resulting in a syscall mismatch (most likely `sys_brk` vs.

[1] `https://github.com/dpoetzsch/asb-examples`

Variant 1	Variant 2
Hashing 0xffffffff	Hashing 0xddddddddd
Hash result = no collision	Hash result= Collision

Diverging Behavior

Table 1: Example of a control flow divergence during a pointer hashing

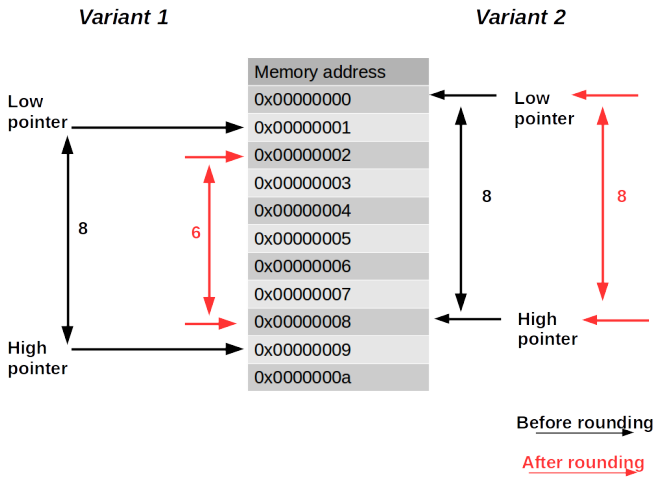

Figure 1: Rounding to 2 in diverse variants

any other syscall). Since we can not predict when a table resize might occur, these mismatches happen randomly. For instance, `xchat_gnome` uses the `HashMgr::add_word()` function to hash objects by using the pointer value (call to `store_pointer()`). The resulting error is a syscall mismatch `sys_brk` vs. `sys_read` that does not occur at every run. Other known occurences of pointer hashing are:

Glibc `G_direct_hash()` in `gutils.c:2093`

Libgtk+2.0 `gtk_rc_styles_hash()` in `gtkrc.c:1967` and `gtk_gc_value_hash()` in `gtkgc.c:282`

libhunspell `HashMgr::add_word()` in `hashmgr.cxx:168`

Other pointer values computation could be address sensitive. LLVM's data flow sanitizer for example, massively uses the `uptr` type which is the integer representation of a pointer. Address sensitivity was found in the `FastPoisonShadow` function in `asan_poisoning.h:39`. This function is part of the memory poisoning process and reserves the memory that will be poisoned. While doing so, the lowest memory address is

rounded up to the page size and the upper one is rounded down. In fig. 1, the rounding only affects variant 1 while variant 2 remains the same, creating a divergence. The size of the required space is later given as an argument to a `map` syscall, resulting in an argument mismatch in the MVEE.

As similar strategy is found in the implementation of `ptmalloc2` which is the default memory allocator in GNU libc. This function requires that all of its arenas (i.e. heaps) are aligned to a specific boundary. To ensure that, `ptmalloc2` allocates a larger than neccessary region and unmaps the lower and upper parts to reach the desired alignment. The size of the memory to unmap being most probably different, this may result in an argument mismatch during the unmap system call. So far, the solution adopted by GHUMVEE is to detect these malloc calls and to temporarily allow this specific diverging syscall pattern.

2.3. Pointer in binary data written using a syscall like `writev`

With layout diversification like ASLR the pointer values change each single run of a program. Thus, in a MVEE each variant will have different pointer values and writing those pointers results in an argument mismatch.

Many X11 programs like `xpdf`, `xedit`, `xpaint`, ... show that behavior. We inspected `xpdf` in detail. `xpdf` uses a library that writes pointers to GUI handler functions into a display buffer which at some point is written out using the syscall `writev`.

To be precise, we only found out that the two variants write those pointers into the display buffer and that, at a later point, the program crashes because of a divergence in the display buffer. We did not check, if the divergence that we found is actually the divergence that makes it crash (there could be more and the one that we found could even be overwritten before the crash). Our best guess is that most widget creations induce divergences in the display buffer and that we only inspected the very first one after the start of the program. However, it seems likely that all divergences are caused by the same address sensitive problem (pointers in the display buffer).

Please refer to appendix B for the technical details on the creation of the inspected divergence in the display buffer.

To get an idea of the extend of the investigated problem we performed a simple `grep` over the sources of the packages `motif`, `libX11`, `libXaw`, `libxcb`, `libXt` and `x11-apps`:

```
$ find . -name *.[hc] -exec grep 'XtSetArg(' {} + \
      | grep '&' | wc -l
```

The results show that there is at least 1157 calls to `XtSetArg` (the root of the inspected divergence) with a pointer value. This does not even include the function pointers as the inspected libraries do not use the address-of operator for getting the function pointers. It seems that a fix for function pointers alone does not suffice to fix the problem.

2.4. Writing a padded structure

One may be tempted to write a whole structure using `write(fd, struct_ptr, nbytes)` as in the `WriteTargetsTable()` function mentioned in appendix B.

However, it is very likely that some padding bytes were added between the structure members due to data alignment. This happens if the total number of bytes required by the members $\neq 0$ *mod* 4. If the structure is placed on the stack, these padding bytes will result in the exact same error as if an uninitialized value was used.

This problem is also unlikely to occur without any code or data layout diversification for the same reason as in the uninitialized value problem. Listing 1 shows a snippet demonstrating this address sensitive behavior.

```
struct padded_struct {
    char ch1; // 1 byte
              // 3 padding bytes
    int i1;   // 4 bytes on 64bits system
    int i2;   // 4 bytes
};

int main(int argc, char *argv[]) {
    struct padded_struct foo;

    foo.ch1 = 'a';
    foo.i1 = 0;
    foo.i2 = 1;

    printf("sizeof␣padded_struct␣is␣%ld\n", sizeof(foo));
      /* sizeof padded_struct is 12 */

    write(2, &foo, sizeof(foo));
      /* Argument mismatch triggered if ASLR is enabled */
    return 0;
}
```

Listing 1: Writing a padded structure

The first way to fix this is to remove the padding bytes by adding the `__packed__` attribute to the structure. Another solution is to set the padding bytes to zero. A simple way to implement is to just use of `memset` after each declaration of the structure. It is clear that the adopted solution strongly depends on the program's architecture. Other implementations of this padding bytes zeroing could be investigated and may be even implemented in the compiler itself. Still, writing a structure will not be safe unless we make sure that no pointer values will be written (see previous category).

3. Detection tools

Address sensitivity sometimes results from bugs or hazardous coding. Such behaviors are often detectable at compilation or run time, given the right tool. We tried the following

tools on our minimal examples of address sensitivity[2]: Valgrind, the `-Wall` flags of `gcc` and `clang`, and the LLVM sanitizers for addresses, memory and dataflow[3].

As expected, the use of uninitialized values is detected by both `gcc`'s and `clang`'s `-Wuninitialized` option. But still, if an array or a structure was not initialized, `clang` would not detect it and `gcc` would report it only if optimizations were turned on. Valgrind remains the best way to detect uses of uninitialized variables at run time.

Regarding structure padding, it appears that `gcc` fails at detecting the use of uninitialized memory. However, `clang`'s memory sanitizer detects it at runtime, just like valgrind would.

We found no tools to detect the problems described in sections 2.2 and 2.3 since these idioms are generally not considered being hazardous. So far, manually debugging these kinds of address sensitive programs implies paying attention to every pointer to integer cast. Sometimes, the cast itself is responsible for the address sensitivity like in hashing functions (`gtk_gc_value()` or `g_direct_hash()`). The first functionnality of such an address sensitivity detection tool would be to detect every integer to pointer cast and then keep track of the casted variable uses.

4. Conclusion

We listed and categorized the adress sensitive behaviors that we encountered while running GHUMVEE. Especially pointer hashing and pointer writing seem to be a common cause of address sensitive behavior. It is worth noting that further research could still reveal other forms of address sensitivity.

For uninitialized values and padded structures we discussed detection tools. Uninitialized values create highly implementation specific (and often even undefined) behavior. Also, they can easily be removed in the cases inspected. Thus, we think that it should be solved in the individual applications. For the problems related to padded structures we discussed solutions in section 2.4.

It remains to investigate solutions for the problems caused by written pointers and treating pointers as integers. For the latter problem, we are looking into an approach that creates a virtual address space that is shared between the variants. Whenever a pointer is cast into an integer, its value would be mapped into this address space. We are still investigating the implications of this approach, but it seems that under certain conditions it could solve the pointer hashing problem.

References

[1] David Chisnall, Colin Rothwell, Robert N. M. Watson, Jonathan Woodruff, Munraj Vadera, Simon W. Moore, Michael Roe, Brooks Davis, and Peter G. Neumann. Beyond the PDP-11: architectural support for a memory-safe C abstract machine. In Özcan Özturk, Kemal Ebcioglu, and Sandhya Dwarkadas, editors, *Proceedings of*

[2] `https://github.com/dpoetzsch/asb-examples`

[3] `-fsanitize=address`, `-fsanitize=memory`, `-fsanitize=dataflow`

the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '15, Istanbul, Turkey, March 14-18, 2015, pages 117–130. ACM, 2015.

[2] Per Larsen, Stefan Brunthaler, and Michael Franz. Automatic software diversity. *IEEE Security & Privacy*, 13(2):30–37, 2015.

[3] Babak Salamat, Todd Jackson, Andreas Gal, and Michael Franz. Orchestra: intrusion detection using parallel execution and monitoring of program variants in user-space. In Wolfgang Schröder-Preikschat, John Wilkes, and Rebecca Isaacs, editors, *Proceedings of the 2009 EuroSys Conference, Nuremberg, Germany, April 1-3, 2009*, pages 33–46. ACM, 2009.

[4] PaX Team. ASLR documentation. `http://pax.grsecurity.net/docs/aslr.txt`.

[5] Stijn Volckaert, Bart Coppens, and Bjorn De Sutter. Cloning your gadgets: Complete rop attack immunity with multi-variant execution. 2015.

Appendices

A. List of tested programs

Programs with no appearent address sensitivity

`bash 4.3.11`, `bzip2 1.0.6`, `cat 8.21`, `chmod 8.21`, `cp 8.21`, `date 8.21`, `df 8.21`, `echo 8.21`, `grep 2.16`, `less 458`, `ls 8.21`, `ssh 6.6.1p1`

Address sensitive programs and libraries

glibc 2.19 Uses an uninitialized value as a source of randomness and pointers as hash keys. See sections 2.1 and 2.2 for details.

gnome-calculator 3.10.3 Does not even run with diversification disabled.

xchat-gnome 0.3.0 Non-deterministically `xchat-gnome` shows different behavior. One recurring problem is the use of pointers as hash keys as described in section 2.2.

xpdf 3.03 Pointer values are written out using the syscall `writev`. See section 2.3 for details.

xedit 7.7+2 Shows similar problems as `xpdf`.

xpaint 2.9.0 Shows similar problems as `xpdf`.

gdb 7.7.1 Does not even run with diversification disabled.

LLVM Memory sanitizer 3.7.0 Rounding poninters for alignment reasons. See section 2.2 for details.

B. Creation of a divergence in the display buffer

Table 2 shows the technical key steps of how the divergence in the display buffer that we inspected is created. The trace starts at the point when different pointers are written into data arrays.

#	Location	Explanation
1	libXm, ScrolledW.c:1433	A function pointer to a GUI handler is written into an `Arg` array using `XtSetArg`
2	libXm, ScrolledW.c:1437	`XmDropSiteRegister` is called which will write the mentioned `Arg` array into the display buffer which will be passed to `writev`
3	libXm, DropSMgr.c:3280	Call `XtGetSubresources` with the `Arg` array and the `fullInfoRec` structure
4	libXt, Resources.c:549	Copy the data from the elements in the `Arg` array into the memory region that corresponds to the `import_targets` field of the `fullInfoRec` structure. There might also be a bug here, we found an offset by 2 for some reason.
5	libXm, DropSMgr.c:3355	Call `_XtTargetsToIndex` with the `fullInfoRec.import_targets` that now contain the function pointers from the `Arg` array
6	libXm, DragBS.c:1393	The `newTargets`, which are a copy of the previous `import_targets`, are appended to the `targetsTable`
7	libXm, DragBS.c:1394	Call to `WriteTargetsTables` which will write the `targetsTable` into the display buffer
8	libXm, DragBS.c:905	`targetsTable` is copied into the `fill` part of the memory region that is prepared to be written into the display buffer
9	libXm, DragBS.c:928	`XChangeProperty` is called which will write the `propertyRecPtr` array (which also contains the `fill` area with the function pointers) into the display buffer (`display->buffer`)

Table 2: Creating a divergence in the display buffer in `xpdf`